Apuntes U 1

Tema ***Análisis de Algoritmos***

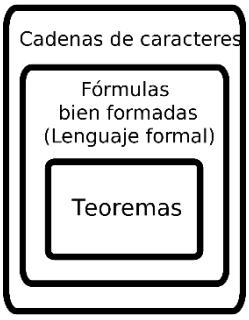
"Hay dos maneras de diseñar software: una es hacerlo tan simple que sea obvia su falta de deficiencias, y la otra es hacerlo tan complejo que no haya deficiencias obvias"
C.A.R. Hoare

¿Qué es un Lenguaje de Programación?

Un lenguaje de programación es un lenguaje formal diseñado para expresar procesos que pueden ser llevados a cabo por máquinas como las computadoras.
En matemáticas, lógica, y ciencias de la computación, un lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados.

¿Qué es un Lenguaje Formal?

En matemáticas, lógica y ciencias de la computación, un lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados. Al conjunto de los símbolos primitivos se le llama el alfabeto (o vocabulario) del lenguaje, y al conjunto de las reglas se lo llama la gramática formal (o sintaxis). A una cadena de símbolos formada de acuerdo a la gramática se la llama una fórmula bien formada (o palabra) del lenguaje. Estrictamente hablando, un lenguaje formal es idéntico al conjunto de todas sus fórmulas bien formadas. A diferencia de lo que ocurre con el alfabeto (que debe ser un conjunto finito) y con cada fórmula bien formada (que debe tener una longitud también finita), un lenguaje formal puede estar compuesto por un número infinito de fórmulas bien formadas.



¿Qué son los paradigmas de Programación?

Un paradigma de programación es una propuesta tecnológica adoptada por una comunidad de programadores cuyo núcleo central es incuestionable en cuanto a que únicamente trata de resolver uno o varios problemas claramente delimitados.
En general, la mayoría de los paradigmas son variantes de los dos tipos principales de programación, imperativa y declarativa. En la programación imperativa se describe paso a paso un conjunto de

instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema.

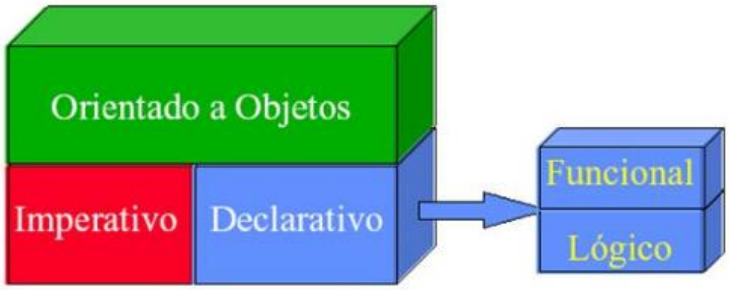
Imperativo o por procedimientos: es considerado el más común y está representado es decir elogiado, por ejemplo, por C, BASIC o Pascal.
Declarativo: por ejemplo, la programación funcional, la programación lógica, o la combinación lógico-funcional.

a. Funcional: está representado por Scheme o Haskell. Este es un caso del paradigma declarativo.
b. Lógico: está representado por Prolog. Este es otro caso del paradigma declarativo.

Orientado a objetos: está representado por Smalltalk, un lenguaje completamente orientado a objetos.

Programación dinámica: está definida como el proceso de romper problemas en partes pequeñas para analizarlos.

Programación multiparadigma: es el uso de dos o más paradigmas dentro de un programa.



¿Qué es un Algoritmo?

El término algoritmo proviene del árabe **al-Khowârizmî**, sobrenombre del célebre matemático árabe **Mohámed ben Musa**. En el diccionario de la Real Academia española se define como el conjunto ordenado y finito de operaciones que permiten hallar la solución a un problema. El término de algoritmo se puede entender como la descripción de cómo resolver un problema. El conjunto de instrucciones que especifican la secuencia de operaciones a realizar, en orden, para resolver un sistema específico o clase de problemas, también se denomina algoritmo. En otras palabras, un algoritmo es una "especie de fórmula" para la resolución de un problema.

Propiedades:

1) **Ser finito**. Un algoritmo tiene que acabar tras un número finito de pasos. El número de pasos de un algoritmo, por grande y complicado que sea el problema que soluciona, debe ser limitado. Todo algoritmo incluye los pasos inicio y fin.

2) **Precisión**. Esto significa que los pasos u operaciones del algoritmo deben desarrollarse en un orden estricto.

3) **Ser definido**. Cada paso de un algoritmo debe de tener un significado preciso; las acciones a realizar han de estar especificadas en cada paso rigurosamente y sin ambigüedad.

El algoritmo se desarrolla como paso fundamental para desarrollar un programa, el computador sólo desarrollará las tareas programadas, con los datos suministrados; no puede improvisar ni inventa el dato que necesite para realizar un paso. Por ello, cuantas veces se ejecute el algoritmo, el resultado depende estrictamente de los datos suministrados, de hecho si se ejecuta con un mismo conjunto de datos de entrada, el resultado será siempre el mismo.

4) **Presentación formal**: el algoritmo debe estar expresado en alguna de las formas comúnmente aceptadas. Las formas de presentación de algoritmos son, entre otras: el pseudocódigo, diagrama de flujo y diagramas de Nassi/Schneiderman.

5) **Conjunto de entradas**. Debe existir un conjunto específico de objetos, cada uno de los cuales constituye los datos iniciales de un caso particular del problema que resuelve el algoritmo. A este conjunto se llama conjunto de entrada del algoritmo.

6) **Conjunto de salidas**. Debe existir un conjunto específico de objetos, cada uno de los cuales constituye la salida o respuesta que debe tener el algoritmo para los diferentes casos particulares del problema. Para cada entrada del algoritmo debe existir una salida asociada.

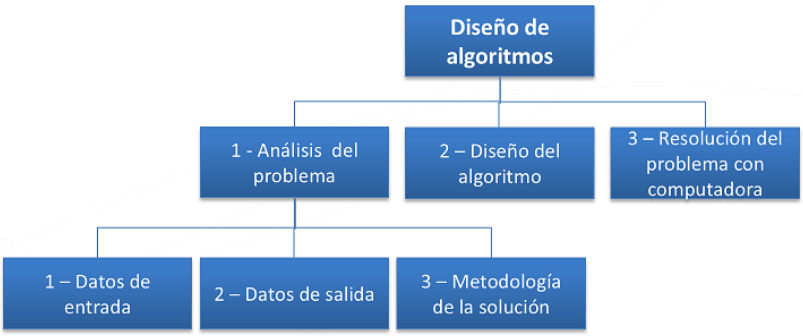
7) **Efectividad (o Corrección)**. El algoritmo debe satisfacer la necesidad o solucionar el problema para el cual fue diseñado. Para garantizar que el algoritmo logre el objetivo, es necesario ponerlo a prueba; a esto se le llama verificación o prueba de escritorio.

8) **Eficiencia**: En cuanto menos recursos requiere será más eficiente el algoritmo. Este concepto se aplica, entre otros puntos, las formas de almacenar los datos, de leerlos, etc.

Datos e información: En el uso diario, datos e información son esencialmente sinónimos. Sin embargo, en las ciencias de la computación suelen hacer una diferencia: datos se refiere a la representación de algún hecho, concepto o entidad real (los datos pueden tomar diferentes formas, por ejemplo, palabras escritas o habladas, números y dibujos); información que implica datos procesados y organizados

Diseño de algoritmos

Los métodos más eficaces para el proceso de diseño se basan en el conocido **divide y vencerás**, es decir, la resolución de un problema complejo se realiza dividiendo el problema en sub-problemas y a continuación dividir estos sub-problemas en otros de nivel más bajo o sea más simples, hasta que pueda ser implementada una solución en la computadora y así sucesivamente (de lo global a lo concreto); es lo que se denomina diseño descendente **Top-Down design o modular**.



Una vez realizado un primer acercamiento al problema, éste se ha de ampliar, (romper el problema en cada etapa y expresar cada paso en forma más detallada) lo que denominamos como **refinamiento del algoritmo o Stepwise Refinement**.

Cada sub-problema es resuelto mediante un **módulo o sub-programa**, que tiene un solo punto de entrada y un solo punto de salida.

Cualquier programa bien diseñado consta de un programa principal (el módulo de nivel más alto) que llama a sub-programas (módulos de nivel más bajo) que a su vez puede llamar a otros sub-programas.
Los programas estructurados de esta forma se dicen que tienen un diseño modular y el método de romper el programa en módulos más pequeños se llama programación modular.
Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente (incluso por diferentes programadores) y a continuación combinarlos entre ellos.
El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina:

1. Programar el módulo.
2. Comprobar el módulo.
3. Si es necesario, depurar el módulo.
4. Combinar el módulo con los módulos anteriores.

El proceso que convierte los resultados del análisis del problema en un diseño modular, con refinamiento sucesivos que permitan una posterior traducción a un lenguaje, se denomina diseño del algoritmo.
En todos los casos, durante el desarrollo de la materia, evitaremos focalizarnos en un lenguaje y de igual manera todo ejemplo o ejercicio que les pida deberá ser resuelto con las instrucciones propias, que veamos dentro de algoritmos. Por ejemplo: Resumiendo todo lo visto hasta ahora, en diseño top-down, modular sería:

Es decir que resolver un problema, implica la resolución de tres sub-problemas en el siguiente orden:

1º. Análisis del problema
2º Diseñar el algoritmo
3º Aplicar la solución en la computadora.

Obviamente cada uno de estos sub-problemas pueden a su vez dividirse en otros; por ejemplo:

Con lo cual vemos que el concepto de dividir un gran problema en pequeños problemas, nos ayuda a:

- Reducir la complejidad para el desarrollo
- Facilitar la comprensión de la lógica de cada proceso
  Cada proceso pasa a tener un objetivo único
- Cualquier cambio se aplica sobre el proceso específico que maneja la situación afectada
- Permite reducir los futuros costos de mantenimiento.
- Permite entregas parciales que ayudan a reducir los tiempos totales.

Dato, Campo, Campo Estructura

Podemos dar como primera definición, que un CAMPO es el elemento que nosotros identificamos mediante un nombre, por lo cual una de las características fundamentales del CAMPO es su nombre; ese nombre que declaramos es el que permite que el algoritmo lo identifique unívocamente y que mediante el mismo lo utilizare en el proceso en las operaciones que desarrollemos. Mientras que el DATO es el valor que ese campo tiene.

Ese valor lo puede obtener desde:

- El exterior

• Por modificaciones durante el proceso.

Por tal motivo consideramos al CAMPO como el nombre y lugar físico en el cual alojaremos el valor o DATO.

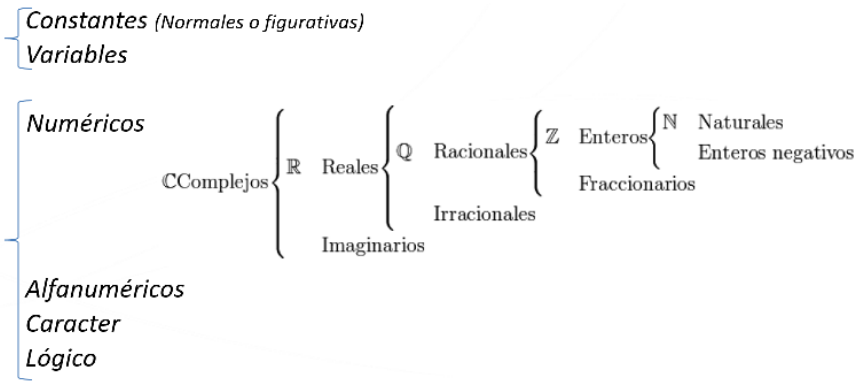
Constante (definición): Un objeto de tipo constante es un valor cuyo contenido no puede variar.
Podríamos diferenciar:

• Constante Normal: Valor constante expresado en si mismo. Por Ej.: Valor numérico 128
• Constante figurativas: Un nombre que de manera figurada simboliza un valor constante que no cambia. Por Ej.: PI = 3,1416; en el que PI sería la constante.

Variable (definición): Lugar en la memoria que está reservado, y que posee un nombre, un tipo y un contenido. El valor de la variable puede cambiar y puede ser modificado a lo largo del programa. La variable tiene un valor determinado, de forma que el tipo de variable, debe de ser uno en concreto, por lo tanto el primer dato de la variable condicionará los posteriores datos que almacenará después.

La variable contiene tres cualidades:
- Nombre
- Tipo
- Valor

Resumiendo
Podemos encontrar solamente 4 tipos de datos en nuestros ejercicios:
• Real
• Entero
• Alfanumérico
• Lógico



Estructura de Datos

Es el conjunto de variables agrupadas y organizadas de cierta forma para representar un información y/o un comportamiento.

En el desarrollo de programas, existe una fase previa a la escritura del programa, esta es el **diseño del algoritmo** que nos guiará a la solución del problema, en esta fase también deberá considerarse la/las estructura/s de datos que se utilizarán.

El término estructura de datos se refiere a la forma en que la información esta organizada dentro de un programa. La correcta organización de datos puede conducir a algoritmos más simples y más eficientes.

Clasificación según su forma de locación

* Estructuras de datos **estáticas**: Son aquellas cuyo tamaño en memoria es fijo, por ejemplo, los arreglos.

* Estructuras de datos **dinámicas**: Son las estructuras que permiten variar su tamaño en memoria de acuerdo a las necesidades del ambiente, por ejemplo, listas enlazadas.

Clasificación según su lugar de locación

* **Lineales**: Son aquellas que se alojan en espacios contiguos de memoria.

* **No Lineales**: Son las estructuras que permiten ser alojadas en espacios no contiguos de memoria.

Estructura de Datos numéricas

1 bit. Bit es el acrónimo Binary digit ('dígito binario'), representa 2 estados y sus valores posibles son 0 o 1

1 nibble = 4 bits. Puede representar todas las combinaciones posibles con 4 bits, o sea 16 valores posibles; de 0 a 16. Se utiliza normalmente para el sistema hexadecimal.

1 Byte = 8 bits. Representa 256 valores posibles, numéricamente del 0 al 255. También se utiliza normalmente para representar un carácter *(Ver código ASCII)*.

1 word = XXBits. En computadoras viejas de arquitectura de 16 bits, un Word era representado por 16 bits, en las actuales de arquitectura de 32 y/o 64 Bits un Word tiene 32 y 64 bits respectivamente. Esto está atado a la tecnología o arquitectura de la computadora ya que es la cantidad de Bits que puede direccionar. También existe el Dword (2 veces el tamaño de un Word)

Kilo	10^3	2^{10}
Mega	10^6	2^{20}
Giga	10^9	2^{30}
Tera	10^{12}	2^{40}
Peta	10^{15}	2^{50}
Exa	10^{18}	2^{60}
Zetta	10^{21}	2^{70}
Yotta	10^{24}	2^{80}
Xona	10^{27}	2^{90}
Weka	10^{30}	2^{100}

Tipos Primitivos. No poseen métodos, no son objetos y no necesitan ser invocados para ser creados.

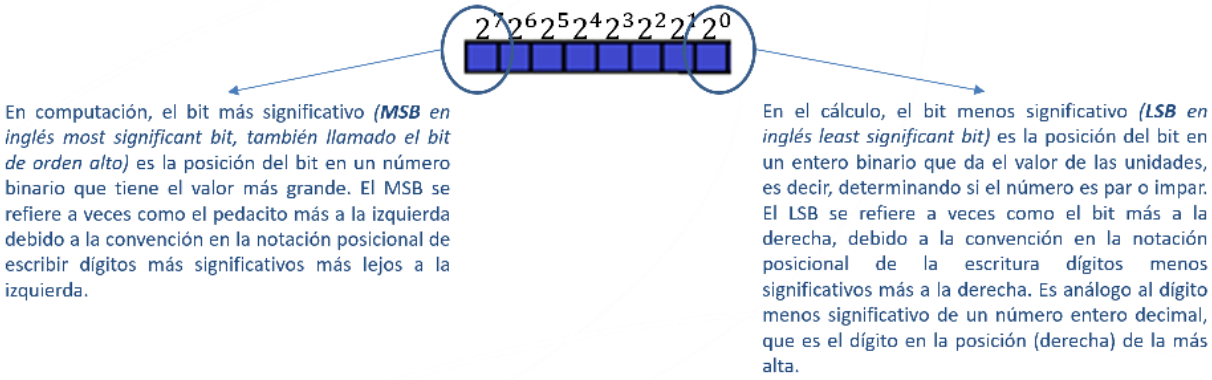
Tipos Objeto. Poseen métodos y necesitan ser invocados (instanciados) para ser creados.

Tipos Primitivos.

NOMBRE	TIPO	OCUPA	RANGO APROXIMADO
byte	Entero	1 byte	-128 a 127
short	Entero	2 bytes	-32768 a 32767
int	Entero	4 bytes	$2*10^9$
long	Entero	8 bytes	Muy grande
float	Decimal simple	4 bytes	Muy grande
double	Decimal doble	8 bytes	Muy grande
char	Carácter simple	2 bytes	---
boolean	Valor true o false	1 byte	---

Estructura "byte" (de 8 bits, 256 números representables)

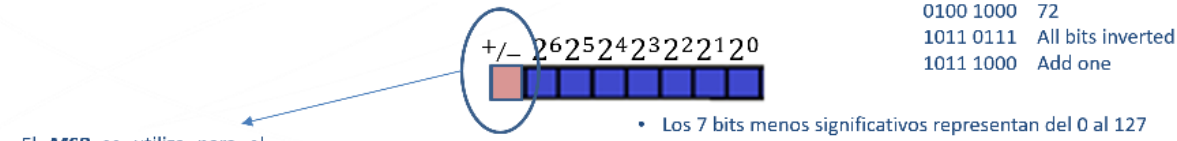
Byte *Unsigned (sin signo),(en c++* ***unsigned char****, en pascal* ***byte****)*:



En computación, el bit más significativo (***MSB*** *en inglés most significant bit, también llamado el bit de orden alto)* es la posición del bit en un número binario que tiene el valor más grande. El MSB se refiere a veces como el pedacito más a la izquierda debido a la convención en la notación posicional de escribir dígitos más significativos más lejos a la izquierda.

En el cálculo, el bit menos significativo *(**LSB** en inglés least significant bit)* es la posición del bit en un entero binario que da el valor de las unidades, es decir, determinando si el número es par o impar. El LSB se refiere a veces como el bit más a la derecha, debido a la convención en la notación posicional de la escritura dígitos menos significativos más a la derecha. Es análogo al dígito menos significativo de un número entero decimal, que es el dígito en la posición (derecha) de la más alta.

Byte *Signed (con signo), (en Java Byte, en c++* ***signed char****, en pascal* ***shortint****)*:

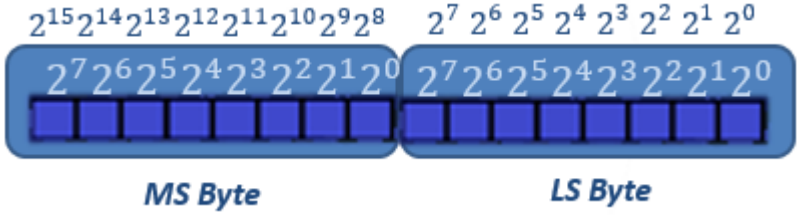


0100 1000 72
1011 0111 All bits inverted
1011 1000 Add one

El ***MSB*** se utiliza para el signo positivo o negativo.

- Los 7 bits menos significativos representan del 0 al 127
- El MSB encendido y el resto apagado, representa el -128
- Existen 128 números negativos (-128 al -1) y 128 números positivos (0 al 127)
- El UB 129 es el SB -127, UB 130 es SB -126, hasta UB 255 que es SB -1.

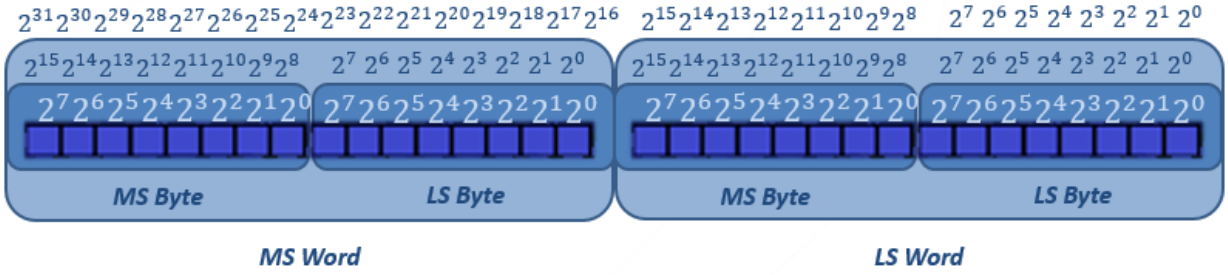
word *Unsigned (En Java no esta definido, en c++* ***unsigned int****, en Pascal* ***word****)*



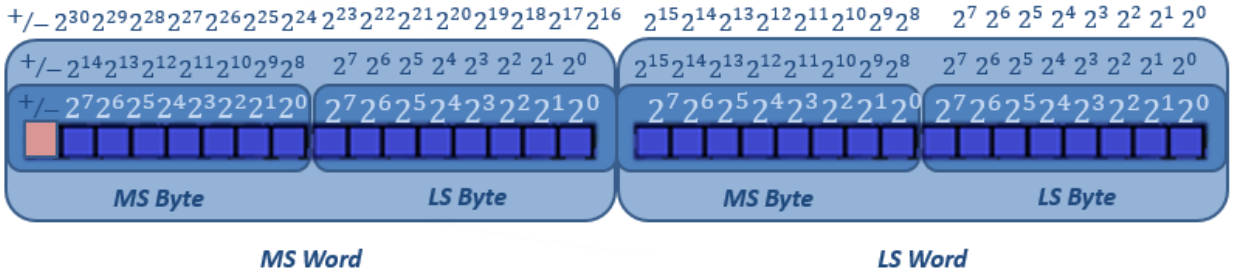
word ***Signed (En Java short, en c++ int, e Pascal integer)***

Estructura “longint o longword” (32 bits o 4 bytes, 4294967295 números representables)

longword *Unsigned (En Java no esta definido, en c++* ***unsigned long int****, en Pascal* ***longword****)*



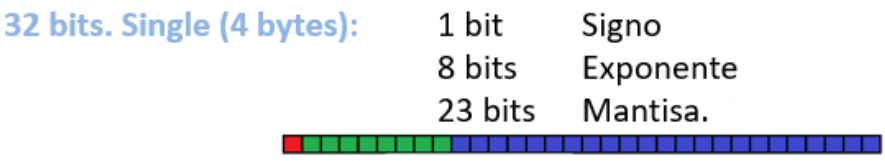
longint *Signed (En Java* ***int****, en c++* ***unsigned long int****, en Pascal* ***longword****)*

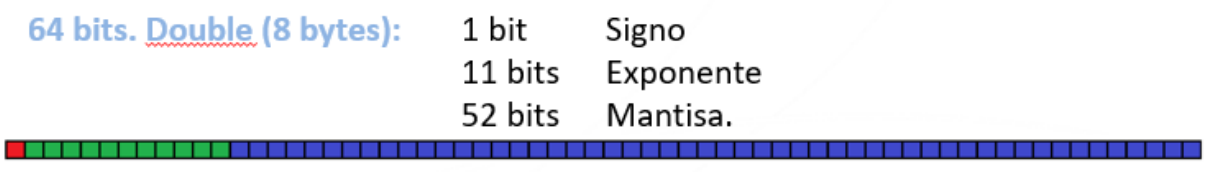


Estructura “float” o coma flotante

Norma IEEE 754, representación de coma flotante:

http://754r.ucbtest.org/standards/754xml.html
http://es.wikipedia.org/wiki/IEEE_coma_flotante





Ejemplo: expresar el número decimal -118,625 usando el sistema IEEE coma flotante.

1) Asignar al bit de signo el valor de "1", dado que el número decimal a convertir es negativo.
2) Transformar en binario -118,625 y queda 1110110,101 *(Se transforma la parte entera a binario. Luego se sigue con la parte fraccionaria, multiplicando cada número por 2. Si el resultado obtenido es mayor o igual a 1 se anota como un uno (1) binario. Si es menor que 1 se anota como un 0 binario. Ej., al multiplicar 0.6 por 2 obtenemos como resultado 1.2 lo cual indica que nuestro resultado es un uno (1) en binario, solo se toma la parte decimal del resultado).*
3) Mover la coma decimal a la izquierda, dejando sólo un 1 a su izquierda: 1110110,101=1,110110101*26. Esto es un número normalizado en coma flotante. El "1" anterior a la coma no se representa en la expresión resultante.
4) Asignar a la mantisa todos los bits a la derecha de la coma decimal, rellenado con ceros a la derecha hasta obtener los 23 bits que ocupa el campo. Es decir, 11011010100000000000000.
5) Sumar al exponente (6) el número 127, convirtiéndose en un exponente desplazado. El total resultante (133) se convierte en un número binario (10000101). El resultado se coloca en el campo "E".

Operaciones fundamentales en algoritmos computacionales.

- *Asignación.*
- *Operaciones Numéricas*
  - *Contadores*
  - *Acumuladores*
- *Operaciones Alfabéticas o Alfanuméricas*
  - *Len*
  - *Concatenación*
- *Operaciones de entrada-salida.*
  - *Entrada*
    - *Ingresar*
    - *Leer*
    - *Eof*
  - *Salida*
    - *Mostrar*
    - *Imprimir*
    - *Escribir*

Representación de Algoritmos.

Pseudocódigo

En ciencias de la computación, y análisis numérico el pseudocódigo (o falso lenguaje) es una descripción de alto nivel compacta e informal1 del principio operativo de un programa informático u otro algoritmo.

Pseudocódigo estilo Pascal:

```
procedimiento bizzbuzz
para i := 1 hasta 100 hacer
    establecer print_number a verdadero;
    Si i es divisible por 3 entonces
        escribir "Bizz";
        establecer print_number a falso;
    Si i es divisible por 5 entonces
        escribir "Buzz";
        establecer print_number a falso;
    Si print_number, escribir i;
    escribir una nueva línea;
fin
```

Pseudocódigo estilo C:

```
subproceso funcion bizzbuzz
para (i <- 1; i<=100; i++) {
    establecer print_number a verdadero;
    Si i es divisible por 3
        escribir "Bizz";
        establecer print_number a falso;
    Si i es divisible por 5
        escribir "Buzz";
        establecer print_number a falso;
    Si print_number, escribir i;
    escribir una nueva línea;
}
```

Utiliza las convenciones estructurales de un lenguaje de programación real2 , pero está diseñado para la lectura humana en lugar de la lectura mediante máquina, y con independencia de cualquier otro lenguaje de programación. Normalmente, el pseudocódigo omite detalles que no son esenciales para la comprensión humana del algoritmo, tales como declaraciones de variables, código específico del sistema y algunas subrutinas. El lenguaje de programación se complementa, donde sea conveniente, con descripciones detalladas en lenguaje natural, o con notación matemática compacta. Se utiliza pseudocódigo pues este es más fácil de entender para las personas que el código del lenguaje de programación convencional, ya que es una descripción eficiente y con un entorno independiente de los principios fundamentales de un algoritmo. Se utiliza comúnmente en los libros de texto y publicaciones científicas que se documentan varios algoritmos, y también en la planificación del desarrollo de programas informáticos, para esbozar la estructura del programa antes de realizar la efectiva codificación.
No existe una sintaxis estándar para el pseudocódigo, aunque los ocho IDE's que manejan pseudocódigo tengan su sintaxis propia. Aunque sea parecido, el pseudocódigo no debe confundirse con los programas esqueleto que incluyen código ficticio, que pueden ser compilados sin errores. Los diagramas de flujo y UML pueden ser considerados como una alternativa gráfica al pseudocódigo, aunque sean más amplios en papel.

El pseudocódigo es una herramienta de programación en la que las instrucciones se escriben en palabras similares al español o inglés, que facilitan tanto la escritura como la lectura de programas, y permite a quien lo realiza concentrarse en las estructuras de control sin tener presente las características propias de algún lenguaje en especial. En esencia se puede definir como un lenguaje de especificaciones de algoritmos.
El pseudocódigo se encuentra a un solo paso de pasarse a un lenguaje de programación y es entendible, de igual forma que los diagramas de flujo, a diferencia que no es gráfico pero igual de comprensible.

Cuenta con palabras reservadas y la forma de su uso, y tiene una estructura mínima que se debe respetar. Siempre la primera palabra debe ser:

Algoritmo: nombre
Seguida de " : " y el nombre que le definamos para el mismo.
Debe tener una sentencia única de FIN, que indica la finalización del proceso.

¿Que es Indentación? Es un anglicismo (de la palabra inglesa *indentation*) de uso común en Informática y significa mover un bloque de texto hacia la derecha insertando espacios o tabuladores para separarlo del texto adyacente, lo que en el ámbito de la imprenta se ha denominado siempre como *sangrado* o sangría.
En los lenguajes de Computación, la indentación se utiliza para mejorar la legibilidad del código por parte de los programadores, teniendo en cuenta que raramente se consideran los espacios en blanco como sentencias de un programa. Sin embargo, en ciertos lenguajes de programación como Haskell, Occa, y Python, la indentación se utiliza para delimitar la estructura del programa permitiendo establecer bloques de código.
Son frecuentes discusiones entre programadores sobre cómo o dónde usar la indentación, si es mejor usar espacios en blanco o tabuladores, ya que cada programador tiene su propio estilo. Vale aclarar que esta palabra no está reconocida por la Real Academia de Lengua, y en su lugar se debería usar SANGRADO, pero por el uso y costumbre que tiene en la jerga informática para nuestro curso la tomamos como válida, por lo que siempre debemos intentar identificar o marcar los bloques de instrucción de esta manera, lo cual ayuda a la legibilidad del código.

Diagramas de flujo

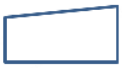
También llamado ordinograma, esta técnica muestra los algoritmos de una manera clara y comprensible.
Es una representación gráfica, es decir, se vale de diversos símbolos para representar las ideas o acciones a desarrollar.
Los diagramas de flujo ayudan en la comprensión de la operación de las estructuras de control. Por ejemplo: el inicio y el fin del algoritmo se representan con un símbolo elíptico, las entradas y salidas con un paralelogramo, las decisiones con un rombo, los procesos con un rectángulo, etc.
Los símbolos utilizados han sido normalizados por el Instituto Norteamericano de Normalización (ANSI) y algunos de ellos son

- *Terminador (inicio o fin)*

- *Conectores*

- *Definición de variables*
- *Asignación de valores a variables*
- *Declaración de expresiones Numéricas*

- *Asignaciones indirectas por teclado de variables en un diagrama de flujo*



- *Visualización por pantalla en un diagrama de flujo*

- *Sub-programa (sub rutina)*

- *Condición lógica*

- *Entrada / Salida*

- *Impresión*

- *Operación de cinta (secuencial)*

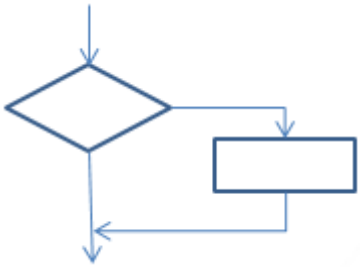
- *Operación de disco (random)*

- *Iteración (for, repetir y mientras)*

Estructuras en un diagrama de Flujo

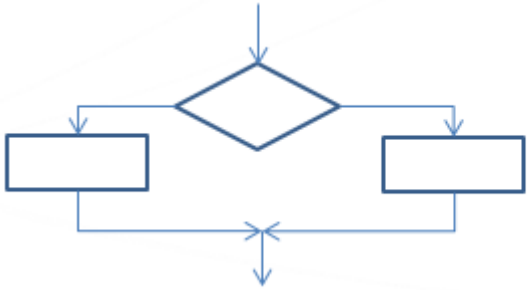
- ***Secuencial**: es la manera natural de organizar las acciones.*

- ***Selectiva simple***



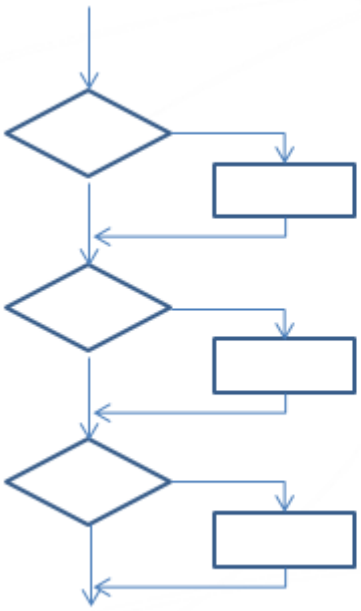
si [condición]
entonces
operación
sino
fin si

- ***Selectiva doble***



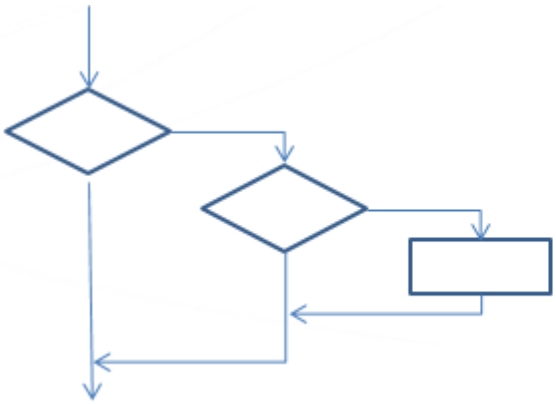
si [condición]
entonces
operación1
sino
operación2
fin si

- ***Selectiva múltiple***



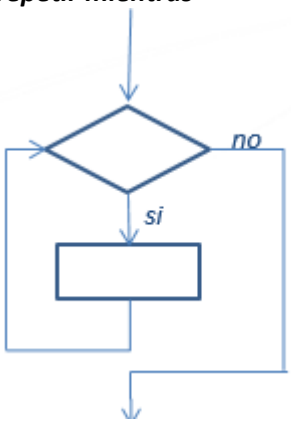
```
si [condición1]
entonces
                operación1
sino
fin si
si [condición2]
entonces
                operación2
sino
fin si
si [condición3]
entonces
                operación3
sino
fin si
```

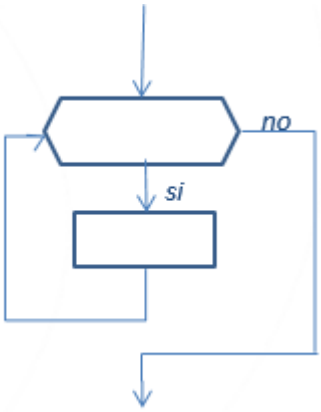
- ***Selectiva anidada***



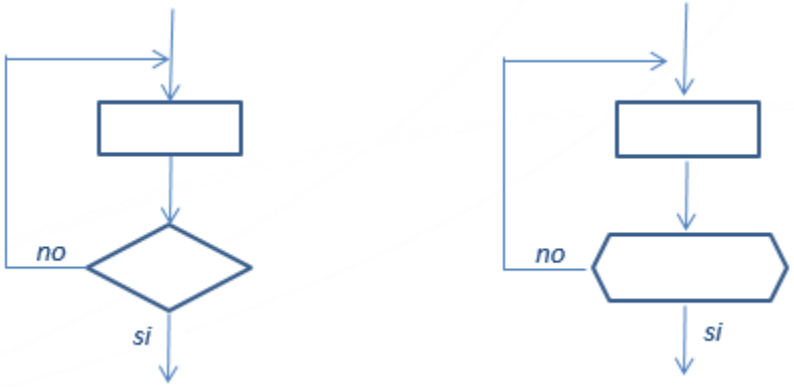
```
si [condición1]
entonces
                si [condición2]
                entonces
                                operación1
                sino
                fin si
sino
fin si
```

- ***Iterativa repetir mientras***





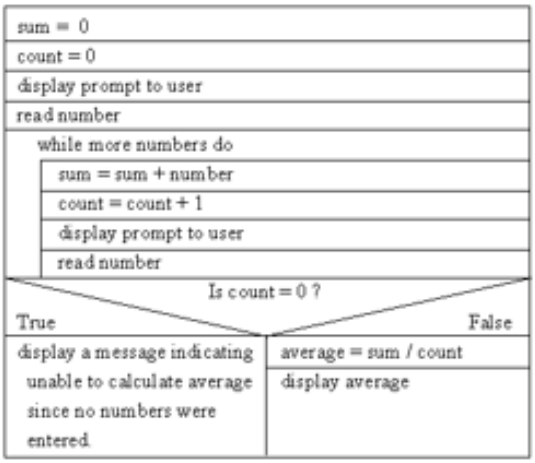
- ***Iterativa repetir hasta***



- ***Iterativa Desde – Hasta*** *(estructura igual al repetir mientras con condiciones distintas)*

Diagramas de Nassi-Shneiderman (o estructograma)

En programación de computadores un diagrama Nassi-Shneiderman (o NSD por sus siglas en inglés), también conocido como diagrama de Chapin1 2 es una representación gráfica que muestra el diseño de un programa estructurado.



Fue desarrollado en 1972 por Isaac Nassi y Ben Shneiderman. Este diagrama también es conocido como estructograma, ya que sirve para representar la estructura de los programas. Combina la descripción textual del pseudocódigo con la representación gráfica del diagrama de flujo.
Basado en un diseño top-down (de lo complejo a lo simple), el problema que se debe resolver se divide en subproblemas cada vez más pequeños - y simples - hasta que solo queden instrucciones simples y construcciones para el control de flujo. El diagrama Nassi-Shneiderman refleja la descomposición del problema en una forma simple usando cajas anidadas para representar cada uno de los subproblemas. Para mantener una consistencia con los fundamentos de la programación estructurada, los diagramas Nassi-Shneiderman no tienen representación para las instrucciones GOTO.
Los diagramas Nassi-Shneiderman se utilizan muy raramente en las tareas de programación formal. Su nivel de abstracción es muy cercano al código de la programación estructurada y ciertas modificaciones requieren que se redibuje todo el diagrama.
Los diagramas Nassi-Shneiderman son (la mayoría de las veces) isomórficos con los diagramas de flujo. Todo lo que se puede representar con un diagrama Nassi-Shneiderman se puede representar con un diagrama de flujo. Las únicas excepciones se dan en las instrucciones GOTO, break y continue.

Expresiones Condicionales.

En una expresión de relación tenemos uno o más valores operados por los operadores correspondientes. Consiste en consecuencia en dos valores operados entre sí, con un operador de relación. Los operadores son:

De relacion.

- *">" Mayor que*
- *"<" Menor que*
- *">=" Mayor o igual que*
- *"<=" Menor o igual que*
- *"=" Igual que*
- *"<>" Distinto que*

Estos operadores, también pueden operar con caracteres (por medio de código ASCII). Ej. :
"A" < "B" Tendría un valor lógico de Verdadero pues el valor ASCII de A es menor que el de B (El código ASCII esta ordenado de acuerdo con el abecedario pero en Inglés, es decir que letras como la Ñ o letras con signos de puntuación se consideran caracteres especiales y se encuentran en otro orden.)
Una expresión relacional obtiene un valor lógico (VERDADERO O FALSO).

Expresiones lógicas

- *NOT (No)*
- *AND (Y)*
- *OR (O)*

A	B	NOT A	A AND B	A OR B
V	V	F	V	V
V	F	F	F	V
F	V	V	F	V
F	F	V	F	F

Operador NOT: Operador que actúa sobre un solo valor. Este operador tiene la cualidad de invertir el valor lógico. Ej. :

NOT <Valor Lógico>
NOT VERDADERO ..... Tendría un valor FALSO
NOT FALSO..... Tendría una valor VERDADERO

Operador AND: La operación AND vale VERDADERO cuando los dos valores son verdaderos, de lo contrario vale FALSO. Ej. :

<Valor Lógico> AND <Valor Lógico>

Operador OR: Es similar a AND en el sentido de que llevará un valor lógico a izquierda y derecha, pero la diferencia estriba en que OR devuelve VERDADERO cuando cualquiera de los dos valores es VERDADERO.

Los operadores relacionales tienen prioridad sobre los lógicos, siempre y cuando ningún paréntesis indique lo contrario. La prioridad de los operadores lógicos es la siguiente:

1. NOT

2. AND
3. OR